Dependency Tree Automata

Stirling, C. (2009). Dependency Tree Automata. In de Alfaro, L. (Ed.), Foundations of Software Science and Computational Structures. (pp. 92-106). (Lecture Notes in Computer Science). Springer Berlin / Heidelberg. doi: 10.1007/978-3-642-00596-1_8

Dependency Tree Automata

Colin Stirling

School of Informatics
Informatics Forum
University of Edinburgh
email: cps@inf.ed.ac.uk

Abstract. We introduce a new kind of tree automaton, a dependency tree automaton, that is suitable for deciding properties of classes of terms with binding. Two kinds of such automaton are defined, nondeterministic and alternating. We show that the nondeterministic automata have a decidable nonemptiness problem and leave as an open question whether this is true for the alternating version. The families of trees that both kinds recognise are closed under intersection and union. To illustrate the utility of the automata, we apply them to terms of simply typed lambda calculus and provide an automata-theoretic characterisation of solutions to the higher-order matching problem.
Keywords: tree automata, binding terms, typed lambda calculus.

1 Introduction

A standard method for solving problems over families of terms is to show that the solutions are tree recognisable: that is, that there is a tree automaton that accepts a term if, and only if, it is a solution to the problem [4]. In such a case, terms are built out of a finite family of (graded) symbols, that is symbols with an arity, which are naturally represented as trees. A tree automaton involves a finite set of states and a finite set of transitions. It traverses a term bottom-up or top-down labelling it with states according to the transitions and if it succeeds then the term is accepted.

Many logical and computational notations employ binders such as $\exists x$, μX, λx, $a(x)$ from first-order logic, fixed-point logic, lambda calculus, π-calculus, and so on. Although each term of such a notation can be represented as a finite tree, to represent families of such terms may require an infinite alphabet: as illustrated by the following formulas $\forall z.\exists f_1 \ldots \exists f_n.f_n(f_{n-1}(\ldots(f_1(z))\ldots))$ for all $n \geq 0$. Although there is research in extending standard automata to infinite alphabets, see the survey [9], it does not cover the specific case caused by binding.

We introduce a new type of tree automaton, a dependency tree automaton, for recognising terms with binding. To maintain a finite alphabet, terms are represented as finite trees which also have an extra binary relation $\downarrow$ between their nodes that represents binding: an idea partly inspired by nested automata which also employ a binary relation $\downarrow$ between nodes representing nesting such as calls and returns [1, 2]. Two kinds of dependency tree automaton are defined, nondeterministic and alternating. We show that the nonemptiness problem, whether

a given automaton accepts at least one tree, is decidable for nondeterministic automata. However, we are unable to show this for the alternating automata and we are also unable to determine whether they are more expressive than nondeterministic automata. The families of trees that both kinds of automata recognise are closed under intersection and union. To illustrate the utility of the automata, we apply them to terms of simply typed lambda calculus and use alternating automata to provide an automata-theoretic characterisation of solutions to the higher-order matching problem.

In Section 2 we define binding trees and the two kinds of dependency tree automaton and show decidability of nonemptiness for the nondeterministic case. We also illustrate how the nondeterministic automata can be used to recognise normal form terms of simply typed lambda calculus of a fixed type. In Section 3, we apply the alternating dependency tree automata to higher-order matching. The proof of characterisation is presented in Section 4.

2 Dependency Tree Automata

In this section we introduce *binding* trees and *dependency* tree automata that operate on them.

Definition 1. *Assume Σ is a finite graded alphabet where each element $s \in \Sigma$ has an arity $ar(s) \geq 0$. Moreover, Σ consists of three disjoint sets Σ_1 that are the binders which have arity 1, Σ_2 are (the bound) variables and Σ_3 are the remaining symbols. A binding Σ-*tree *is a finite tree where each node is labelled with an element of Σ together with a binary relation $\downarrow$ (representing binding). If node n in the tree is labelled with s and $ar(s) = k$ then n has precisely k successors in the tree, the nodes $n1, \ldots, nk$. Also, if a node n is labelled with a variable in Σ_2 then there is a unique node b labelled with a binder occurring above n in the tree such that $b \downarrow n$. For ease of exposition we also assume the following restrictions on Σ-trees: if node n is labelled with a binder then $n1$ is labelled with an element of $\Sigma_2 \cup \Sigma_3$ and if n is labelled with an element of $\Sigma_2 \cup \Sigma_3$ and ni is a successor then it is labelled with a binder. Let T_Σ be the set of binding Σ-trees.*

Definition 2. *A dependency Σ-tree automaton $\mathsf{A} = (Q, \Sigma, q_0, \Delta)$ where Q is a finite set of states, Σ is the finite alphabet, $q_0 \in Q$ is the initial state and Δ is a finite set of transition rules each of which has one of the following three forms.*

1. *$qs \Rightarrow (q_1, \ldots, q_k)$ where $s \in \Sigma_2 \cup \Sigma_3$, $ar(s) = k$, $q, q_1, \ldots, q_k \in Q$*
2. *$qs \Rightarrow q's'$ where $s \in \Sigma_1$, $s' \in \Sigma_3$ and $q, q' \in Q$*
3. *$(q', q)s \Rightarrow q_1 x$ where $s \in \Sigma_1$, $x \in \Sigma_2$ and $q', q, q_1 \in Q$*

Definition 3. *A run of $\mathsf{A} = (Q, \Sigma, q_0, \Delta)$ on $t \in \mathsf{T}_\Sigma$ is a $(\Sigma \times Q)$-tree whose nodes are pairs (n, q) where n is a node of t and $q \in Q$ labelled (s, q) if n is labelled s in t which is defined top-down with root (ϵ, q_0) where ϵ is the root of t. Consider a node (n, q) labelled (s, q) of a partial run tree which does not have successors. If $s \in \Sigma_2 \cup \Sigma_3$ and $qs \Rightarrow (q_1, \ldots, q_k) \in \Delta$ then the successors of (n, q) are the nodes (ni, q_i), $1 \leq i \leq k$. If $s \in \Sigma_1$, $n1$ is labelled $s' \in \Sigma_3$ and*

$qs \Rightarrow q's' \in \Delta$ then $(n1, q')$ is the successor of (n, q). If $s \in \Sigma_1$, $n1$ is labelled $x \in \Sigma_2$, $m \downarrow n1$ in t, (m, q') occurs above or at (n, q) and $(q', q)s \Rightarrow q_1 x \in \Delta$ then $(n1, q_1)$ is the successor of (n, q). A accepts the Σ-tree t iff there is a run of A on t such that if (n, q) is a leaf then n is a leaf of t. Let $\mathsf{T}_\Sigma(\mathsf{A})$ be the set of Σ-trees accepted by A.

A dependency tree automaton A has a finite set of states Q and transitions Δ (which can be nondeterministic). A run of A on a Σ-tree t adds an additional Q labelling to (a subtree of) t: so it is a $(\Sigma \times Q)$-tree. It starts with (ϵ, q_0) where ϵ is the root of t and q_0 is the initial state of A. Subsequent nodes are derived by percolating states down t. The state at a node that is labelled with a variable not only depends on the state of its immediate predecessor but also on the state of the node that labels its binder. This introduces non-local dependence in the automaton (hence the name). A run on t is accepting if it is complete in the sense that each node of t is labelled with an element of Q: if (n, q) is a leaf of the run tree then n is a leaf of t.

Dependency tree automata were partly inspired by nested word and tree automata [1, 2] which are also an amalgam of a traditional automaton and a binary relation $\downarrow$ on nodes of the (possibly infinite) word or tree. However, in that setting $\downarrow$ represents *nesting* such as provided by bracketing and useful for modelling procedure calls and returns. Nesting involves natural restrictions on the relation $\downarrow$ such as "no-crossings": if $m_1 \downarrow m_2$ and $n_1 \downarrow n_2$ and m_1 is above n_1 then either m_2 is above n_1 or n_2 is above m_2. Such restrictions are not appropriate when modelling binding, for instance as with a formula $\forall f.\exists x.\phi(f(x))$.

A fundamental exemplar of binding is terms of the simply typed lambda calculus. Simple types are generated from a single base type $\mathbf{0}$ using the binary $\rightarrow$ operator[1]: $A \rightarrow B$ is the type of functions from A to B. Assuming $\rightarrow$ associates to the right, if type $A \neq \mathbf{0}$ then it has the form $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow \mathbf{0}$, written $(A_1, \ldots, A_n, \mathbf{0})$ here. The *order* of $\mathbf{0}$ is 1 and the *order* of $(A_1, \ldots, A_n, \mathbf{0})$ is $k+1$ where k is the maximum of the orders of the A_is.

Terms of the simply typed λ-calculus (in Church style) are built from a countable set of typed variables $x, y, \ldots$ and constants $a, f, \ldots$ (each variable and constant has a unique type).

Definition 4. *The smallest set T of simply typed terms is:*

1. *if x (f) has type A then $x : A \in T$ ($f : A \in T$),*
2. *if $t : B \in T$ and $x : A \in T$ then $\lambda x.t : A \rightarrow B \in T$,*
3. *if $t : A \rightarrow B \in T$ and $u : A \in T$ then $(tu) : B \in T$.*

The order of a typed term is the order of its type. In a sequence of unparenthesized applications, we adopt the usual convention that application associates to the left, so $tu_1 \ldots u_k$ is $((\ldots(tu_1)\ldots)u_k)$. The usual definitions of free and bound variable occurrences and when a typed term is closed are assumed. Moreover, we

[1] For simplicity, we assume just one base type: everything that is to follow can be extended to the case of arbitrary many base types.

assume the standard definitions and properties of α-equivalence, β-reduction, η-reduction and $\beta\eta$-equivalence, $=_{\beta\eta}$, such as strong normalization of β-reduction: see, for instance, Barendregt [3].

Fact 1 *Every simply typed λ-calculus term is $\beta\eta$-equivalent to a unique term (up to α-equivalence) in η-*long form *as follows,*

1. *if $t : \mathbf{0}$ then it is $u : \mathbf{0}$ where u is a constant or a variable, or $u\, t_1 \dots t_k$ where $u : (B_1, \dots, B_k, \mathbf{0})$ is a constant or a variable and each $t_i : B_i$ is in η-long form,*
2. *if $t : (A_1, \dots, A_n, \mathbf{0})$ then t is $\lambda y_1 \dots y_n.t'$ where each $y_i : A_i$ and $t' : \mathbf{0}$ is in η-long form.*

Throughout, we write $\lambda z_1 \dots z_m$ for $\lambda z_1 \dots \lambda z_m$. A term is in *normal form* if it is in η-long form.

Definition 5. *For any type A and set of constants C, $T_A(C)$ is the set of closed terms in normal form of type A whose constants belong to C.*

Example 1. The monster type $M = ((((\mathbf{0}, \mathbf{0}), \mathbf{0}), \mathbf{0}), \mathbf{0}, \mathbf{0})$ has order 5. Assume $x_1 : (((\mathbf{0}, \mathbf{0}), \mathbf{0}), \mathbf{0})$, $x_2 : \mathbf{0}$ and $z_i : (\mathbf{0}, \mathbf{0})$ for $i \geq 1$. The following family of terms in normal form $\lambda x_1 x_2.x_1(\lambda z_1.x_1(\lambda z_2 \dots x_1(\lambda z_n.z_n(z_{n-1}(\dots z_1(x_2)) \dots)) \dots))$ for $n \geq 0$ belong to $T_M(\emptyset)$. Even to write down this subset of terms up to α-equivalence requires an alphabet of unbounded size. More technically, M is known not to be finitely generable [6]. However, there is a straightforward representation of this family of terms as binding Σ-trees (when dummy λs are added to fulfil the restrictions in Definition 1). Nodes are labelled with binders $\lambda x_1 x_2$, λz, λ, or with variables x_1, z of arity 1 and x_2 of arity 0: in linear form $\lambda x_1 x_2.x_1(\lambda z.x_1(\lambda z \dots x_1(\lambda z.z(\lambda.z(\dots \lambda.z(\lambda.x_2)) \dots)) \dots))$ where there is an edge $\downarrow$ from the node labelled $\lambda x_1 x_2$ to each node labelled x_1 or x_2, and an edge $\downarrow$ from the first node labelled λz to the last node labelled z, and so on. There are no edges $\downarrow$ from nodes labelled with the empty binder λ. Given such a representation of normal form terms in $T_M(\emptyset)$, dependency tree automata can be defined that recognize subsets: there is a simple deterministic two state automaton that recognizes the subset which have an even number of occurrences of x_1. □

Fact 2 *For any type A and finite C, there is a finite Σ such that every $t \in T_A(C)$ up to α-equivalence is representable as a binding Σ-tree (with dummy λs).*

The nonemptiness problem for classical (bottom-up or top-down) nondeterministic tree automata, whether a given automaton accepts at least one tree, is decidable in linear time. Also, the set of families of trees that are recognizable is regular (which implies closure under complement and intersection) [4].

Theorem 1. *Assume A, A_1 and A_2 are dependency Σ-tree automata.*

1. *The nonemptiness problem, given A is $\mathsf{T}_\Sigma(\mathsf{A}) \neq \emptyset$?, is decidable.*
2. *Given A_1 and A_2, there is an A such that $\mathsf{T}_\Sigma(\mathsf{A}) = \mathsf{T}_\Sigma(\mathsf{A}_1) \cap \mathsf{T}_\Sigma(\mathsf{A}_2)$.*

3. Given A_1 *and* A_2*, there is an* A *such that* $\mathsf{T}_\Sigma(\mathsf{A}) = \mathsf{T}_\Sigma(\mathsf{A}_1) \cup \mathsf{T}_\Sigma(\mathsf{A}_2)$.

Proof. Assume $\mathsf{A} = (Q, \Sigma, q_0, \Delta)$ is a Σ-tree automaton and $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ where Σ_1 are the binders, Σ_2 the variables and Σ_3 the other symbols. Let $\| t \|$ be the height of the Σ-tree t and $|S|$ be the size of the finite set S. We show that if $\mathsf{T}_\Sigma(\mathsf{A}) \neq \emptyset$ then A accepts a Σ-tree t such that $\| t \| \leq (|\Sigma_1||Q|+1)(|\Sigma||Q|+1)$. If A accepts t and $\| t \| > l(|\Sigma||Q|+1)$ then in the accepting run of A on t there are l nodes of t, $n_1, \ldots, n_l$ with the same label in Σ and labelled with the same state $q \in Q$ such that each n_i occurs above n_j when $i < j$. Let $B(i,j)$, where $1 \leq i < j \leq l$, be the set of pairs binders $a \in \Sigma_1$ and states $q' \in Q$ such that there is a node n' between n_i and n_j (excluding n_j) labelled with a and q' in the successful run of A on t such that there is an edge $n' \downarrow n''$ where n'' is n_j or occurs below it in t. Also, let $U(i)$ be the set of pairs binders $a \in \Sigma_1$ and states $q \in Q$ such that there is a node n' above n_i in t labelled with a and q in the successful run of A on t. Clearly, if $B(i,j) \subseteq U(i)$ then there is a smaller Σ-tree t' which is accepted by A: the subtree at node n_i is replaced with the subtree at n_j and any edge $n' \downarrow n''$ where n'' is n_j or below it and n' is between n_i and n_j (excluding n_j) is replaced with an edge $n \downarrow n''$ where n is the node above n_i labelled with the same binder and state as n'. Clearly, if A accepts t then it accepts t'. By simple counting, there must be an i, j with $1 \leq i < j \leq (|\Sigma_1||Q|+1)$ such that $B(i,j) \subseteq U(i)$. Therefore, nonemptiness is decidable. The other parts of the theorem follow from the usual product and disjoint union of automata (which here includes the binding relations). □

We do not know if the families of trees recognized by these automata are closed under complement.

Definition 6. *An alternating dependency* Σ*-tree automaton* $\mathsf{A} = (Q, \Sigma, q_0, \Delta)$ *is as in Definition 2 except for the first clause for transitions which now is*

1. $qs \Rightarrow (Q_1, \ldots, Q_k)$ *where* $s \in \Sigma_2 \cup \Sigma_3$*,* $ar(s) = k$*,* $q \in Q$ *and* $Q_1, \ldots, Q_k \subseteq Q$*.*

Definition 7. *A run of alternating dependency* Σ*-automaton* $\mathsf{A} = (Q, \Sigma, q_0, \Delta)$ *on* $t \in \mathsf{T}_\Sigma$ *is a* $(\Sigma \times Q)$*-tree whose nodes are pairs* (n, α) *where* n *is a node of* t *and* $\alpha \in Q^*$ *is a sequence of states, labelled* (s, q) *if* n *is labelled* s *in* t *and* $\alpha = \alpha' q$ *which is defined top-down with root* (ϵ, q_0) *where* ϵ *is the root of* t*. Consider a node* (n, α) *labelled* (s, q) *of a partial run tree which does not have successors. If* $s \in \Sigma_2 \cup \Sigma_3$ *and* $qs \Rightarrow (Q_1, \ldots, Q_k) \in \Delta$ *then the successors of* (n, α) *are* $\{(ni, \alpha q') \mid 1 \leq i \leq k \text{ and } q' \in Q_i\}$*. If* $s \in \Sigma_1$*,* $n1$ *is labelled* $s' \in \Sigma_3$ *and* $qs \Rightarrow q's' \in \Delta$ *then* $(n1, \alpha q')$ *is the successor of* (n, α)*. If* $s \in \Sigma_1$*,* $n1$ *is labelled* $x \in \Sigma_2$*,* $m \downarrow n1$ *in* t*,* $(m, \alpha' q')$ *occurs above or at* (n, α) *and* $(q', q)s \Rightarrow q_1 x \in \Delta$ *then* $(n1, \alpha q_1)$ *is the successor of* (n, α)*.* A *accepts the* Σ*-tree* t *iff there is a run of* A *on* t *such that if* $(n, \alpha q)$ *is a leaf labelled* (s, q) *of the run tree then either* s *has arity* 0 *or* $qs \Rightarrow (\emptyset, \ldots, \emptyset) \in \Delta$*. Let* $\mathsf{T}_\Sigma(\mathsf{A})$ *be the set of* Σ*-trees accepted by* A*.*

A run of an alternating automaton on a Σ-tree t is itself a tree built out of the nodes of t and sequences of states Q^+. There can be multiple copies of nodes of t

within a run because a transition applied to a node $n\ qs \Rightarrow (Q_1, \ldots, Q_k)$ spawns individual copies at ni for each state in Q_i. These automata are alternating as the Q_is can be viewed as conjuncts $\bigwedge_{q \in Q_i} q$ and nondeterminism provides the disjuncts.

Classically, nondeterministic and alternating tree automata accept the same families of trees and the nonemptiness problem for alternating automata is decidable in exponential time [4]. We do not know whether nondeterministic dependency tree automata are as expressive as the alternating automata. Also, it is an open question whether the nonemptiness problem for alternating dependency tree automata is decidable. However, the families of trees recognized by the alternating automata are closed under intersection and union using a similar argument to Theorem 1.

Despite these open expressiveness and algorithmic questions, we shall show that alternating dependency tree automata do have an interesting application.

3 Application of Dependency Automata

We apply alternating dependency tree automata to higher-order matching.

Definition 8. *A matching problem in simply typed lambda calculus is an equation $v = u$ where $v, u : \mathbf{0}$ are in normal form and u is closed. The order of the problem is the maximum of the orders of the free variables $x_1, \ldots, x_n$ in v. A solution is a sequence of terms $t_1, \ldots, t_n$ such that $v\{t_1/x_1, \ldots, t_n/x_n\} =_{\beta\eta} u$ where $v\{t_1/x_1, \ldots, t_n/x_n\}$ is the simultaneous substitution of t_i for each free occurrence of x_i in v for $i : 1 \leq i \leq n$.*

Given a matching problem $v = u$, one question is whether it has a solution: can v be pattern matched to u? The motivation here is a different question: is there an automata-theoretic characterization of the set of solutions of a matching problem? Comon and Jurski define (almost classical) bottom-up tree automata that characterize all solutions to a 4th-order problem [5]: the slight elaboration is the use of $\Box_A$ symbols standing for arbitrary typed subterms of type A. The authors describe two problems with extending their automata beyond the 4th-order case. The first is how to guarantee only finitely many states. States of their automata are constructed out of observational equivalence classes of terms due to Padovani [8]. Up to a 4th-order problem, one only needs to consider finitely many terms. With 5th and higher orders, this is no longer true and one needs to quotient the potentially infinite terms into their respective observational equivalence classes in order to define only finitely many states: however as Padovani shows this procedure is, in fact, equivalent to the matching problem itself [8]. The second problem is how to guarantee that the alphabet has finite size. As we saw with the monster type in Example 1, fifth-order terms may (essentially) contain infinitely many different variables. In [14], we overcame the first problem but not the second: relative to a fixed finite alphabet, the set of solutions over that alphabet to a matching problem is tree automata recognizable. The proof relies on a similar technology to that used here (a game-theoretic characterisation of

matching). We now overcome the second problem using alternating dependency tree automata.

Definition 9. *Assume $u : \mathbf{0}$ and $w : A$ are closed terms in normal form and $x : (A, \mathbf{0})$. An interpolation problem P has the form $xw = u$. The type of problem P is that of x and the order of P is the order of x. A solution of P of type B is a closed term $t : B$ in normal form such that $tw =_\beta u$. We write $t \models P$ if t is a solution of P.*

Because terms are in η-long form, β-equality and $\beta\eta$-equality coincide (for instance, see [15] for a recent account).

Conceptually, interpolation is simpler than matching because there is a single variable x that appears at the head of the equation. If $v = u$ is a matching problem with free variables $x_1 : A_1, \ldots, x_n : A_n$ where v and u are in normal form, then its *associated* interpolation problem is $x(\lambda x_1 \ldots x_n.v) = u$ where $x : ((A_1, \ldots, A_n, \mathbf{0}), \mathbf{0})$. This appears to raise order by 2 as with the reduction of matching to pairs of interpolation equations in [10]. However, we only need to consider potential solution terms (in normal form with the right type) $\lambda z.zt_1 \ldots t_n$ where each $t_i : A_i$ is closed and so cannot contain z: we say that such terms are *canonical*.

Proposition 3. *A matching problem has a solution iff its associated interpolation problem has a canonical solution.*

Proof. Assume $v = u$ is a matching problem with $x_1 : A_1, \ldots, x_n : A_n$ as free variables and where v and u are in normal form. If it has a solution $t_1, \ldots, t_n$ where each t_i is in normal form, then $v\{t_1/x_1, \ldots, t_n/x_n\} =_\beta u$. Clearly, it therefore follows that $\lambda z.zt_1 \ldots t_n(\lambda x_1 \ldots x_n.v) =_\beta v\{t_1/x_1, \ldots, t_n/x_n\} =_\beta u$. Conversely, if $\lambda z.zt_1 \ldots t_n$ is a canonical solution to its associated interpolation problem $x(\lambda x_1 \ldots x_n.v) = u$ then $t_1, \ldots, t_n$ solves the problem $v = u$. □

In the literature there are slight variant definitions of matching. Statman describes the problem as a range problem [11]: given $v : (A_1, \ldots, A_n, B)$ and $u : B$ where both u and v are closed, are there terms $t_1 : A_1, \ldots, t_n : A_n$ such that $vt_1 \ldots t_n =_{\beta\eta} u$? If $B = (A_1, \ldots, A_m, \mathbf{0})$ is of higher type then u in normal form is $\lambda x'_1 \ldots x'_m.w$. Therefore, we can consider the matching problem $(vx_1 \ldots x_n)c_1 \ldots c_m = w\{c_1/x'_1, \ldots, c_m/x'_m\}$ where the c_i's are new constants that cannot occur in a solution term. In [8] a matching problem is a family of equations $v_1 = u_1, \ldots, v_m = u_m$ to be solved uniformly: they reduce to a single equation $fv_1 \ldots v_m = fu_1 \ldots u_m$ where f is a constant of the appropriate type.

Example 2. The matching problem $x_1(\lambda z.x_1(\lambda z'.za)) = a$ from [5] is 4th-order where $z, z' : (\mathbf{0}, \mathbf{0})$ and $x_1 : (((\mathbf{0}, \mathbf{0}), \mathbf{0}), \mathbf{0})$. Its associated interpolation problem is $x(\lambda x_1.x_1(\lambda z.x_1(\lambda z'.za))) = a$ with $x : (((((\mathbf{0}, \mathbf{0}), \mathbf{0}), \mathbf{0}), \mathbf{0}), \mathbf{0})$. A canonical solution has the form $\lambda x.x(\lambda y.y(\lambda y_1^1 \ldots y(\lambda y_1^k.s)\ldots))$ where s is the constant a or one of the variables y_1^j, $1 \leq j \leq k$. □

Definition 10. *If P is $xw = u$ then C_P is the set of constants that occur in u together with one fresh constant $b : \mathbf{0}$.*

Fact 4 *Let C be any set of constants and let P be an interpolation problem of type B. If $t \models P$ and $t \in \mathsf{T}_B(C)$ then there is a $t' \in \mathsf{T}_B(C_p)$ such that $t' \models P$.*

Given a potential solution term t in normal form to the interpolation problem P, $xw = u$, there is the tree in Figure 1. If $x : (A, \mathbf{0})$ then the explicit application

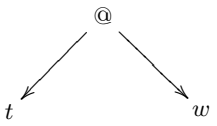

Fig. 1. An interpolation tree

operator @ : $((A, \mathbf{0}), A, \mathbf{0})$ has its expected meaning: $@tw = tw$. Our goal is to define an alternating dependency tree automaton that accepts the tree in Figure 1 when t is a solution of P. By Fact 4 we can assume that any such solution term t only contains constants in the finite set C_P. Moreover, we assume the following representation of binders and variables. A binder $\lambda\overline{y}$ is such that either $\overline{y}$ is empty and therefore is a dummy λ and can not bind a variable occurrence or $\overline{y} = y_1 \dots y_k$ and $\lambda\overline{y}$ can only then bind variable occurrences of the form y_i, $1 \leq i \leq k$. Consequently, in the binding tree representation if $n \downarrow m$ and n is labelled $\lambda y_1 \dots y_k$ then m is labelled y_i for some i.

In general the right term u of an interpolation problem may contain bound variables: for instance, $x(\lambda z.z) = f(\lambda x_1 x_2 x_3.x_1 x_3)a$ has order 3 where x has type $((\mathbf{0}, \mathbf{0}), \mathbf{0})$ and $f : (((\mathbf{0}, \mathbf{0}), \mathbf{0}, \mathbf{0}, \mathbf{0}), \mathbf{0}, \mathbf{0})$ assuming $x_2 : \mathbf{0}$. For ease of exposition, as it simplifies the presentation considerably, we restrict ourselves to the case where there are no such variables: this is discussed further in Section 5.

Definition 11. *Assume $u : \mathbf{0}$ is closed and does not contain bound variables. The set of subterms of u, $Sub(u)$, is defined inductively: if $u = a : \mathbf{0}$ then $Sub(u) = \{u\}$ and if $u = f u_1 \dots u_k$ then $Sub(u) = \bigcup_{1 \leq i \leq k} Sub(u_i) \cup \{u\}$.*

Given P, we assume a simple finite alphabet Σ (containing C_P, the constants in w, @ and suitable $\lambda\overline{y}$s and variable occurrences).

Example 3. In the case of Example 2, there is the finite syntax where $\Sigma_1 = \{\lambda x, \lambda y, \lambda y', \lambda x_1, \lambda z, \lambda z', \lambda\}$, $\Sigma_2 = \{x, y, y', x_1, z, z'\}$ and $\Sigma_3 = \{a, b, @\}$. □

The states of our dependency tree automaton are based on Ong [7] (which is a different setting, with a fixed infinite λ-term built out of a fixed finite alphabet and an alternating parity automaton). To give intuition, consider a game-theoretic understanding (such as with game-semantics) of Figure 1 where $t = \lambda z.z t_1 \dots t_n$ and $w = \lambda\overline{x}.w'$ as pictured in Figure 2. In the game, play jumps around the interpolation tree. It starts at @ and proceeds down from λz to z and then jumps to $\lambda\overline{x}$ of w (as it labels the subterm that would replace z in a β-reduction). It then proceeds down w and eventually may reach x_j, in which case

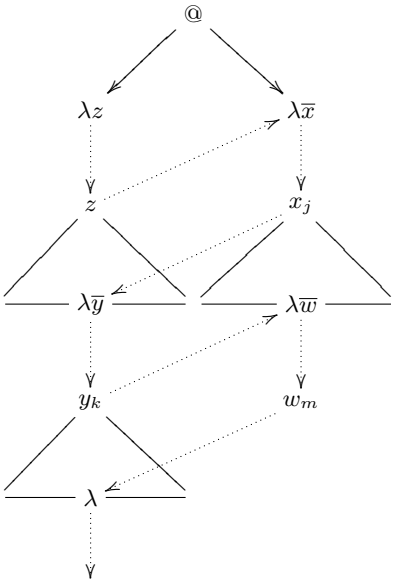

Fig. 2. Game-theoretic view

it jumps to the jth successor of z in t labelled with $\lambda\overline{y}$ and then play proceeds in t and may eventually reach y_k and so jump to the kth successor of x_j in w and so on. The question is how to capture jumping within a tree automaton. This we do using variable assumptions as in [14]: Ong calls them "variable profiles" in his setting.

Definition 12. *Assume Σ is the alphabet for P, $xw = u$, and $\mathsf{R} = Sub(u)$. Relative to R, for each variable $z \in \Sigma_2$ the set of z assumptions, $\Gamma(z)$ is defined inductively: if $z : \mathbf{0}$ then $\Gamma(z) = \{(z, r, \emptyset) \mid r \in \mathsf{R}\}$; if $z : (A_1, \ldots, A_k, \mathbf{0})$ then $\Gamma(z) = \{(z, r, \Gamma) \mid r \in \mathsf{R}, \Gamma \subseteq \bigcup_{1 \leq i \leq k} \bigcup_{x : A_i \in \Sigma_2} \Gamma(x)\}$. A mode is a pair (r, Γ) where $r \in \mathsf{R}$ and $\Gamma \subseteq \bigcup_{z \in \Sigma_2} \Gamma(z)$.*

A variable assumption is an abstraction from a sequence of moves in a game. For instance $(z, u, \{(x_j, r_1, \{(y_k, r_2, \{(w_m, r_3, \emptyset)\})\})\})$ abstracts from the play pictured in Figure 2: the subterms r_i of u represent what is left of u that still needs to be achieved in order for $tw =_\beta u$.

A mode is a pair (r, Γ) where $r \in \mathsf{R}$ and Γ is a set of variable assumptions. Because R is finite and Σ is fixed, there can only be boundedly many different modes (r, Γ): modes are the states of our automaton.

Definition 13. *Assume P is $xw = u$, Σ is its alphabet and $\mathsf{R} = Sub(u)$. The dependency tree automaton is $\mathsf{A}_P = (Q, \Sigma, q_0, \Delta)$ where Q is the set of modes (r_i, Γ_i), $q_0 = (u, \emptyset)$ and the transition relation Δ is defined on nodes of the binding Σ-tree by cases on Σ.*

1. $(u, \emptyset)@ \Rightarrow (\{(u, \Gamma)\}, \{(u, \Gamma_1)\})$ *if* $\Gamma = \{(z, u, \Gamma_1)\}$
2. $((r, \Gamma), (r', \Gamma'))\lambda\overline{y} \Rightarrow (r', \Sigma)x_i$ *if* $(x_i, r', \Sigma) \in \Gamma$
3. $(fr_1 \dots r_k, \Gamma)\lambda\overline{y} \Rightarrow (fr_1 \dots r_k, \emptyset)f$
4. $(a, \emptyset)\lambda\overline{y} \Rightarrow (a, \emptyset)$
5. $(r, \Gamma)x_j \Rightarrow (Q_1, \dots, Q_k)$ *if* $Q_i = \{(r', \Gamma') \,|\, (y_i, r', \Gamma') \in \Gamma\}$ *for each* $i : 1 \leq i \leq k$ *and* $ar(x_j) = k > 0$
6. $(fr_1 \dots r_k, \emptyset)f \Rightarrow (\{(r_1, \emptyset)\}, \dots, \{(r_k, \emptyset)\})$

The root of the interpolation tree labelled @ has two successors t of the form $\lambda z.zt_1 \dots t_n$ and w. The automaton starts with state $(u, \emptyset)$ at @ and then a single z variable assumption (z, u, Γ_1) is chosen. The state at node labelled λz is then $(u, \{(z, u, \Gamma_1)\})$ and (u, Γ_1) at the other successor of @. Assume the current state is (r', Γ') at node n of the interpolation tree labelled $\lambda\overline{y}$. If $n1$ is labelled with variable x_i and $m \downarrow n1$ then m is labelled $\lambda x_1 \dots x_k$ for some k and the state above (r', Γ') at m has the form (r, Γ) where Γ is a set of assumptions for each x_j, $1 \leq j \leq k$. One of the x_i assumptions, (x_i, r', Σ) where the right term r' is as in the state at n is chosen and state (r', Σ') labels $n1$. If $n1$ is labelled f then for the automaton to proceed from node n to $n1$, r' must have the form $fr_1 \dots r_k$. In which case $n1$ is labelled with state $(r', \emptyset)$. Similarly, if $n1$ is labelled with the constant $a : \mathbf{0}$ then r' must be a and $\Gamma' = \emptyset$. If the state is (r, Γ) at node n of the interpolation tree and n is labelled x_j with arity $k > 0$ then Γ consists of sets of y_i assumptions, $1 \leq i \leq k$ for some y (reflecting when play would return to successors of n: for instance, in Figure 2 play jumps from x_j to $\lambda\overline{y}$ and returns to x_js kth successor if it reaches y_k and there can be multiple occurrences of y_k meaning that there could be further returns jumps). For each y_i assumption (y_i, r', Γ') the automaton spawns a copy at ni with state (r', Γ'). Finally, if the state is $(fr_1 \dots r_k, \emptyset)$ at node n of the interpolation tree labelled with f then the automaton proceeds down each successor ni with state $(r_i, \emptyset)$.

Theorem 2. *Assume P is $xw = u$, Σ is the alphabet and A_P is the dependency Σ-tree automaton in Definition 13. For any canonical Σ-term t, A_P accepts the tree @tw iff $t \models P$.*

4 Proof of Theorem 2

The proof of Theorem 2 employs a game-theoretic interpretation of an interpolation tree as illustrated in Figure 2 and developed in [14]. (It avoids questions, answers and justification pointers of game-semantics [7] and uses iteratively defined look-up tables.)

Assume P is the problem $xw = u$, Σ is the alphabet, $\mathsf{R} = \mathrm{Sub}(u)$ and t is a potential solution term. We define the game $\mathsf{G}(t, P)$ played by one participant, player $\forall$, the *refuter* who attempts to show that t is not a solution of P. The game is played on the Σ-binding tree @tw of Figure 1.

Definition 14. *N is the set of nodes of the binding tree @tw labelled with elements of $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$ and S is the set $\{[\,x\,] \,|x \in \mathsf{R} \cup \{\forall, \exists\}\}$ of game-states.*

$[\forall]$ *and* $[\exists]$ *are the final game-states. Let* N_1 *be the subset of nodes* N *whose labels belong to* Σ_1 *(the binders). For each* $i \geq 1$*, the set of look-up tables* Θ_i *is iteratively defined:* $\Theta_1 = \{\theta_1\}$ *where* $\theta_1 = \emptyset$ *and* Θ_{i+1} *is the set of partial maps from* $N_1 \to (\bigcup_{s\in\Sigma_1} N^{ar(s)} \times \bigcup_{j\leq i} \Theta_j)$*.*

Definition 15. *A play of* $\mathsf{G}(t, P)$ *is a finite sequence* $n_1q_1\theta_1, \ldots, n_lq_l\theta_l$ *of positions where each* $n_i \in N$*, each* $q_i \in S$ *and* q_l *is final and each* $\theta_i \in \Theta_i$ *is a look-up table. For the initial position* n_1 *is the root of the interpolation tree labelled* @*,* $q_1 = [\, u \,]$ *where* u *is the right term of* P *and* θ_1 *is the empty look-up table. Player* $\forall$ *loses the play if the final state is* $[\exists]$*, otherwise she wins the play.*

The game $\mathsf{G}(t, P)$ appeals to a finite set of states S comprising goal states $[\, r \,]$, $r \in \mathsf{R}$, and *final* states, $[\forall]$, winning for the refuter, and $[\exists]$, losing for the refuter. The central feature of a play of $\mathsf{G}(t, P)$, as depicted in in Figure 2, is that repeatedly control may jump from a node of t to a node of w and back again. Therefore, as play proceeds, one needs an account of the meaning of free variables in subtrees. A free variable in a subtree of t (a subtree of w) is associated with a subtree of w (a subtree of t). This is the role of the look-up table $\theta_k \in \Theta_k$ at position $k \geq 1$. If n is labelled $\lambda y_1 \ldots y_m$ and $\theta_k(n)$ is defined then it is of the form $((n_1, \ldots, n_m), \theta_j)$ which tells us that any node m labelled y_i such that $n \downarrow m$ is associated with the subtree rooted at node n_i: that subtree may itself contain free variables, hence, the presence of a previous look-up table θ_j.

Current position is $n[r]\theta$. Next position by cases on label at node n:

1. @ then $n1[r]\theta'$ where $\theta' = \theta\{((n2), \theta)/n1\}$
2. $\lambda\overline{y}$ then $n1[r]\theta$
3. $a : \mathbf{0}$ if $r = a$ then $n[\exists]\theta$ else $n[\forall]\theta$
4. $f : (B_1, \ldots, B_k, \mathbf{0})$ if $r = fr_1 \ldots r_k$ then $\forall$ chooses $j \in \{1, \ldots, k\}$ and $nj\,[r_j]\theta$ else $n[\forall]\theta$
5. $y_j : \mathbf{0}$ if $m \downarrow n$ and $\theta(m) = ((m_1, \ldots, m_l), \theta')$ then $m_j\,[r]\theta'$
6. $y_j : (B_1, \ldots, B_k, \mathbf{0})$ if $m \downarrow n$ and $\theta(m) = ((m_1, \ldots, m_l), \theta')$ then $m_j\,[r]\theta''$ where $\theta'' = \theta'\{((n1, \ldots, nk), \theta)/m_j\}$

Fig. 3. Game moves

Definition 16. *If the current position in* $\mathsf{G}(t, P)$ *is* $n[r]\theta$ *and* $[r]$ *is not final then the next position is determined by a unique move in Figure 3 according to the label at node* n*.*

At the initial node labelled @, play proceeds to its first successor labelled λz and the look-up table is updated as its other successor is associated with λz. Later, if play reaches a node labelled z (bound by initial successor of root) then it jumps to the second successor of the root node. Standard updating notation is assumed: $\gamma\{((m1, \ldots, mk), \gamma')/n\}$ is the partial function similar to γ except

that $\gamma(n) = ((m1, \ldots, mk), \gamma')$ where n will be labelled $\lambda y_1 \ldots y_k$ for some y. If play is at a node labelled $\lambda\overline{y}$, where $\overline{y}$ can be empty, then it descends to its successor. At a node labelled with the constant $a : \mathbf{0}$, the refuter loses if the goal state is $[\, a \,]$ and wins otherwise. At a node labelled with a constant f with arity more than 0, $\forall$ immediately wins if the goal state is not of the form $[\, fr_1 \ldots r_k \,]$. Otherwise $\forall$ chooses a successor j and play moves to its jth successor. If play is at node n labelled with variable $y_j : \mathbf{0}$ and $\theta(m) = ((m_1, \ldots, m_l), \theta')$ when $m \downarrow n$ then play jumps to m_j and θ' becomes the look-up table. If n is labelled $y_j : (B_1, \ldots, B_k, \mathbf{0})$ and $\theta(m) = ((m_1, \ldots, m_l), \theta')$ when $m \downarrow n$ then play jumps to m_j which is labelled $\lambda x_1 \ldots x_k$ for some x and the look-up table is θ' together with the association of $((n1, \ldots, nk), \theta)$ to m_j.

Definition 17. *Player $\forall$ loses the game $\mathsf{G}(t, P)$ if she loses every play of it and otherwise she wins the game.*

Lemma 1. *Player $\forall$ loses $\mathsf{G}(t, P)$ iff $t \models P$.*

Proof. Given $P : A$, $xw = u$ and $t : A$ either $t \models P$ or $t \not\models P$. Because the simply typed λ-calculus is strongly normalizing, it follows that there is an m such that tw reduces to normal form using at most m β-reductions (whatever the reduction strategy). For any position $n[r]\theta$ of a play of $\mathsf{G}(t, P)$ we say that it m-holds (m-fails) if $r = \exists$ ($r = \forall$) and when not final, by cases on the label at n(where look-up tables become delayed substitutions and we elide between nodes, subtrees and terms)

- @ then $n1n2 =_\beta r$ ($n1n2 \neq_\beta r$) and $n1n2$ normalizes with m β-reductions
- λ then $n1\theta =_\beta r$ ($n1\theta \neq_\beta r$) and $n1\theta$ normalizes with m β-reductions
- $\lambda y_1 \ldots y_k$ then $n1\theta =_\beta r$ ($n1\theta \neq_\beta r$) and $n1\theta$ normalizes with $(m - k)$ β-reductions
- f then $n\theta =_\beta r$ ($n\theta \neq_\beta r$) and $n\theta$ normalizes with m β-reductions
- $y_j : \mathbf{0}$ if $n' \downarrow n$ and $\theta(n') = ((n_1, \ldots, n_l), \theta')$ then $n_j\theta' =_\beta r$ ($n_j\theta' \neq_\beta r$) and $n_j\theta'$ normalizes with m β-reductions
- $y_j : (B_1, \ldots, B_k, \mathbf{0})$ if $n' \downarrow n$ and $\theta(n') = ((n_1, \ldots, n_l), \theta')$ then $t' =_\beta r$ ($t' \neq_\beta r$) where $t' = (n_j\theta')n1\theta \ldots nk\theta$ and t' normalizes with m β-reductions

Initially, play is at n labelled @ with state [u] and the empty look-up table: therefore, as either $tm =_\beta u$ or $tm \neq_\beta u$ it follows that for some m, either $n[u]\theta_1$ m-holds or m-fails. The following invariants are easy to show by case analysis.

1. If $n[r]\theta$ m-holds (m-fails), n labels $\lambda y_1 \ldots y_k$ and $n'[r']\theta'$ is the next position then it $(m - k)$-holds ($(m - k)$-fails)
2. If $n[r]\theta$ m-holds (m-fails), n labels λ and $n'[r']\theta'$ is the next position then it m-holds (m-fails)
3. If $n[r]\theta$ m-holds and n labels f and $n'[r']\theta'$ is any next position then it m'-holds for $m' \leq m$
4. If $n[r]\theta$ m-fails and n labels f then some next position $n'[r']\theta'$ m'-fails for some $m' \leq m$

5. If $n[r]\theta$ m-holds (m-fails) and n labels y_j and $n'[r']\theta'$ is the next next position then it m-holds (m-fails)

From these invariants it follows first that if a non-final position m-holds then any next position m'-holds for some $m' \leq m$ and second if a non-final position $n[r]\theta$ m-fails then there is a next position that m'-fails for some $m' \leq m$. Moreover, there cannot be an infinite sequence of positions (as the index m strictly decreases with a move at a node labelled $\lambda y_1 \ldots y_k$, $k > 0$, and must be 0 at a node labelled with a constant $a : \mathbf{0}$). Therefore, the result follows. □

In the following we let $p \in \mathsf{G}(t, P)$ abbreviate that p is a position in some play of $\mathsf{G}(t, P)$. If such a position p is at a node labelled with a variable then we identify the earlier position at the node labelled with its binder when the value of that binder in the look-up table at p is defined.

Definition 18. *Assume $p_1 = n_1 q_1 \theta_1, \ldots, p_l = n_l q_l \theta_l$ is a play of $\mathsf{G}(t, P)$ and n_j is labelled with a variable. Position p_i is a parent of p_j iff $n_i \downarrow n_j$ and $\theta_i(n_i) = \theta_j(n_i)$.*

Fact 5 *If $p \in \mathsf{G}(t, P)$ is at a node labelled with a variable then there is a unique $q \in \mathsf{G}(t, P)$ that is the parent of p.*

We now extend the notion of a successor in a tree to positions in a play.

Definition 19. *Assume $p_1 = n_1 q_1 \theta_1, \ldots, p_l = n_l q_l \theta_l$ is a play of $\mathsf{G}(t, P)$, node n_m is a successor of n_k (so, for some j, $n_m = n_k j$) and $1 \leq k < m < l$. Position p_m succeeds position p_k if either $m = k + 1$ or n_k is labelled @, or n_k is labelled with a variable and p_{k+1} is the parent of p_{m-1}.*

Proposition 6. *Assume $p_1 = n_1 q_1 \theta_1, \ldots, p_l = n_l q_l \theta_l$ is a play of $\mathsf{G}(t, P)$ and p_m is a position with $m < l$. There is a unique subsequence of positions $p_{i_1}, \ldots, p_{i_k}$ such that $i_1 = 1$, $i_k = m$ and for all $j : 1 \leq j < k$ position $p_{i_{j+1}}$ succeeds p_{i_j} and for any c if $n_{i_c} \downarrow n_{i_j}$ then p_{i_c} is the parent of p_{i_j}.*

Proof. Assume that $p_1 = n_1 q_1 \theta_1, \ldots, p_l = n_l q_l \theta_l$ is a play of $\mathsf{G}(t, P)$ and p_m is a position with $m < l$. Consider the branch of the interpolation tree from the root labelled @ to n_m. We now pick out the subsequence of positions at these nodes backwards starting with p_m at n_m. Suppose $p_{i_{j+1}}$ is given. If $n_{i_{j+1}}$ is labelled x_i, f or a then p_{i_j} is $p_{i_{j+1}-1}$. If n_{i_j} is labelled $\lambda \overline{y}$ and its immediate predecessor is f then p_{i_j} is also $p_{i_{j+1}-1}$. If n_{i_j} is labelled $\lambda \overline{y}$ and its immediate predecessor is x_i and p_l is the parent of $p_{i_{j+1}-1}$ then p_{i_j} is p_{l-1}. The argument that if $n_{i_c} \downarrow n_{i_j}$ then p_{i_c} is the parent of p_{i_j} is also straightforward. □

Definition 20. *If $p = n[r]\theta \in \mathsf{G}(t, P)$ and n is labelled y_j then its associated variable assumption, $V(p)$, is defined by induction on the type of y_j. If $y_j : \mathbf{0}$ then $V(p) = (y_j, r, \emptyset)$. If $y_j : (B_1, \ldots, B_k, \mathbf{0})$ and p' is the next position after p then $V(p) = (y_j, r, \Gamma)$ where $\Gamma = \{V(q) \mid q \in \mathsf{G}(t, P)$ and p' is the parent of $q\}$.*

Definition 21. *If $p = n[r]\theta \in \mathsf{G}(t, P)$ then $M(p)$ is the mode at node n associated with p defined by cases on the label at n (and which uses Definition 20). If $@$, f or a then $M(p) = (r, \emptyset)$. If y_j and $V(p) = (y_j, r, \Gamma)$ then $M(p) = (r, \Gamma)$. If $\lambda\overline{y}$ then $M(p) = (r, \Gamma)$ where $\Gamma = \{V(q) \mid q \in \mathsf{G}(t, P)$ and p is the parent of $q\}$.*

Theorem 2 is a corollary (via Lemma 1) of the following result.

Theorem 3. *Assume P is $xw = u$, Σ is the alphabet and A_P is the dependency Σ-tree automaton in Definition 13. For any canonical Σ-term t, $\forall$ loses $\mathsf{G}(t, P)$ iff A_P accepts the tree $@tw$.*

Proof. Assume $\forall$ loses $\mathsf{G}(t, P)$. We show that there is a successful run of A_P on $@tw$ via Proposition 6 and Definition 21. More precisely, the successful run tree is built in such a way that for any of its nodes $(n, \alpha(r, \Gamma))$ there is a play $p_1 = n_1 q_1 \theta_1, \ldots, p_l = n_l q_l \theta_l$ of $\mathsf{G}(t, P)$ and a position p_m with $m < l$ such that if $p_{i_1}, \ldots, p_{i_k}$ is the subsequence identified in Proposition 6 then the branch from the root to $(n, \alpha(r, \Gamma))$ consists of nodes $n'_1, \ldots, n'_k$ where $n'_j = (n_{i_j}, M(p_{i_1}) \ldots M(p_{i_j}))$, $1 \leq j \leq k$. Initially this is true as $(n_1, (u, \emptyset))$ is the root node of the run tree when $n_1[u]\theta_1$ is the initial position (of any play). It is now an easy exercise to show that there is always an application of a transition rule of A_P of Definition 13 to a nonterminal node $(n, \alpha(r, \Gamma))$ that preserves this property.

For the other direction assume $\forall$ wins $\mathsf{G}(t, P)$ but there is a successful run of A_P on $@tw$. There is a winning play $p_1 = n_1[r_1]\theta_1, \ldots, p_l = n_l[r_l]\theta_l$ of $\mathsf{G}(t, P)$ for $\forall$. So, n_{l-1} is labelled $a : \mathbf{0}$ or $f : (B_1, \ldots, B_k, \mathbf{0})$ and $r_{l-1} \neq a$ or $r_{l-1} \neq f r_1 \ldots r_k$ because $r_l = \forall$. Let p_m be the earliest position in this play such that there are positions $p_{i_1}, \ldots, p_{i_k}$ of Proposition 6 such that there is a branch of the successful run tree of A_P on $@tw$ consisting of nodes $n'_1, \ldots, n'_{k-1}$ with $n'_{i_j} = (n_{i_j}, \alpha_j(r_{i_j}, \Gamma_j))$ for some α_j and Γ_j, $1 \leq j < k$ but no successor of n'_{k-1} of the form $(n_m, \alpha(r_m, \Gamma))$. We know that there is such a position p_m, $1 < m < l$, because the root of the run tree has the form $(n_1, (r_1, \emptyset))$ and by the transition rules 3 and 4 of Definition 13 there cannot be a node of a successful run tree $(n_{l-1}, \alpha(r_{l-1}, \Gamma))$ for any α and Γ. A case analysis on the label at node n_m shows that if there is such a position p_m then there is an even earlier position with this property which is a contradiction. □

5 Conclusion

We introduced nondeterministic and alternating dependency tree automata for recognising terms with binding. Decidability of nonemptiness is shown for the nondeterministic automata. There are significant open questions for the alternating automata: are they more expressive than the nondeterministic automata and is their nonemptiness problem decidable? We also provided an application of the alternating automata to characterise solutions to a higher-order matching problem. We need to see if there are other applications of these automata.

To save space, we assumed that a right term u in an interpolation problem does not contain bound variables. We handle them as in [14, 13] by including new corresponding constants which are not allowed to occur in solution terms. If u is $f(\lambda x_1 x_2 x_3.x_1 x_3)a$ then c_1, c_2 and c_3 are included where each c_i has the same type as x_i. Definition 11 is refined to only allow closed subterms of base type by replacing bound variables by their corresponding constants: for u above we include a, $c_1(c_3)$ and c_3. A new kind of variable assumption is included, a triple of the form (z_i, r, c) where c is one of the new constants and look-up tables are extended to include entries of the form $\theta_m(z) = c$. Transition rules for the automaton and the game moves are extended accordingly. For instance, in 4 of Figure 3 there is also the case when $r_j = \lambda x_1 \dots x_m.r'$ and n_j is labelled $\lambda y_1 \dots y_m$: so the next position is $nj[r'\{c_1/x_1, \dots, c_m/x_m\}]\theta'$ where $\theta' = \theta\{c_1/y_1, \dots, c_m/y_m\}$.

References

1. Alur, R. and Madhusudan P. Adding nested structure to words *Lecture Notes in Computer Science*, **4036**, 1-13, (2006).
2. Alur, R., Chaudhuri S. and Madhusudan, P. Languages of nested trees, *Lecture Notes in Computer Science*, **4144**, 329-342, (2006).
3. Barendregt, H. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol 2*, ed. Abramsky, S., Gabbay, D. and Maibaum, T., Oxford University Press, 118-309, (1992).
4. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S. and Tommasi, M. *Tree Automata Techniques and Applications*. Draft Book, http://l3ux02.univ-lille3.fr/tata/ (2002).
5. Comon, H. and Jurski, Y. Higher-order matching and tree automata. *Lecture Notes in Computer Science*, **1414**, 157-176, (1997).
6. Joly, T. The finitely generated types of the lambda calculus. *Lecture Notes in Computer Science*, **2044**, 240-252, (2001).
7. Ong, C.-H. L. On model-checking trees generated by higher-order recursion schemes, *Procs LICS 2006*, 81-90. (Longer version available from Ong's web page, 55 pages preprint, 2006.)
8. Padovani, V. Decidability of fourth-order matching. *Mathematical Structures in Computer Science*, **10(3)**, 361-372, (2001).
9. Segoufin, L. Automata and logics for words and trees over an infinite alphabet. *Lecture Notes in Computer Science*, **4207**, 41-57, (2006).
10. Schubert, A. Linear interpolation for the higher-order matching problem. *Lecture Notes in Computer Science*, **1214**, 441-452, (1997).
11. Statman, R. Completeness, invariance and λ-definability. *The Journal of Symbolic Logic*, **47**, 17-26, (1982).
12. Stirling, C. Higher-order matching and games. *Lecture Notes in Computer Science*, **3634**, 119-134, (2005).
13. Stirling, C. A game-theoretic approach to deciding higher-order matching. *Lecture Notes in Computer Science*, **4052**, 348-359, (2006).
14. Stirling, C. Higher-order matching, games and automata. *Procs LICS 2007*, 326-335.
15. Støvring, K. Higher-order beta matching with solutions in long beta-eta normal form. *Nordic Journal of Computing*, **13**, 117-126, (2006).